Using Streaming SIMD Extensions 2 (SSE2) in Motion Compensation for Video Decoding and Encoding

Version 2.0

07/00

Order Number: 248607-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Update for Pentium® 4 processor	07/00
1.0	Original publication of document	09/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. Information Technology - Generic Coding of Moving Pictures and Associated Audio Information, International Standard ISO/IEC DIS 13818-2, Part 2: Video, 1994.
2. MPEG Video Compression Standard, Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. LeGall, C&H, ITP, 1996, pages 28-29, 237-262.
3. Image and Video Compression Standards, Algorithms and Architectures. Vasudev Bhaskaran and Konstantinos Konstantinides, Kluwer Academic Publishers, 1995, pages 91-92, 171-174.
4. Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback, Intel Application Note, AP-529. http://developer.intel.com/drg/mmx/appnotes/ap529.htm.

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) instructions introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced with the Streaming SIMD Extensions (SSE). The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note presents the Motion Compensation (MC) algorithm that is used in the Moving Picture Expert Group (MPEG) video decoding and encoding processes, and includes code examples that exploit the SSE2 instructions to perform this algorithm.

A previous application note, AP-529, "Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback" proposed the use of MMX technology functionality for motion compensation algorithms and the resulting performance gain over equivalent scalar code. This application note describes how Intel architecture, SSE2 instructions, and Pentium 4 processor architecture provide a significant performance gain above and beyond MMX technology.

2 Motion Compensation Algorithm

This section briefly discusses the Motion Compensation (MC) algorithm. The reader is encouraged to use the references listed in the Reference section of this document to study the MPEG compression and the MC concept and algorithm in greater detail. Section 2 in AP-529 provides a short description of the MPEG frame structure. Reference [2] provides a wide background and specifications of the whole MPEG-2 compression standard.

The following section is based on reference [2] and reference [3]. The frames in a video sequence are made up of areas that show little change from frame to frame. Video compression schemes such as MPEG employ inter-frame coding to take advantage of the high degree of similarity between pictures. MC is defined as “the process of compensating for the displacement of moving objects from one frame to another” [3], p. 91. The basic unit to which MC is applied is defined at the macroblock level. A macroblock is built of one luminance component of 16x16 pixels (also known as the Y component), and two chrominance components of 8x8 pixels each (Cb and Cr components).

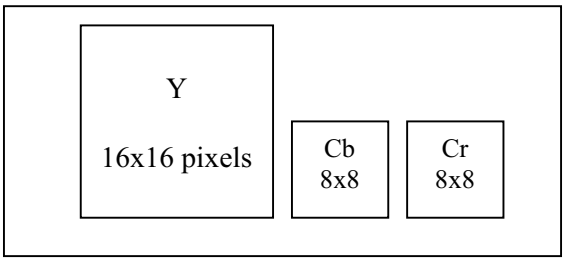

Figure 1: Macroblock Structure – 16x16 Y, 8x8 Cb, and 8x8 Cr Components

To perform MC on a given macroblock in the current frame, we use one or two precalculated reference frames and corresponding motion vectors (offsets). The motion vectors indicate the location of the reference macroblock relative to the current macroblock. MC is performed either by simply copying the reference macroblock, or by interpolating the reference macroblock's neighboring pixels using averaging to produce a smooth transition.

Averaging neighboring pixels (also called half-pel prediction) of a reference macroblock can be done either on:

- two horizontally adjacent pixels
- two vertically adjacent pixels
- four neighboring pixels

When using two separate reference macroblocks, an additional average between the two resultant macroblocks is performed.

The same motion vectors and motion modes (e.g. half-pel direction type) are used for both Y plane and Cb and Cr planes. The code example in this application note presents only the computation of Y pixels; however, a similar code can be used for calculating Cb and Cr pixels.

Until the advent of the SSE2 instructions and SSE instructions, MC was complex and carried a large memory overhead.

The SSE instructions and the SSE2 instructions significantly reduce computation complexity through a special averaging instruction `pavgb`, and by increasing the SIMD parallelism from 8 elements to 16 elements.

MC is a memory intensive module because it refers to one or two macroblocks from frames that are no longer present in the processor caches. Using prefetch instructions may significantly reduce this memory overhead.

2.1 Applications for Motion Compensation

The Motion Compensation (MC) algorithm is used in video compression of moving pictures, including standards such as MPEG (MPEG-1, MPEG-2 and MPEG-4) and H.261. This application note specifically refers to its usage in the MPEG-2 decoding and encoding.

The MC algorithm is a key module in MPEG-2 decoding, and takes about 14-20% of the decoder's execution time. It also exists in MPEG-2 encoding, but consumes relatively less execution time.

The following block diagrams show the relationship between the MC algorithm and other algorithms in the MPEG encoding and decoding processes.

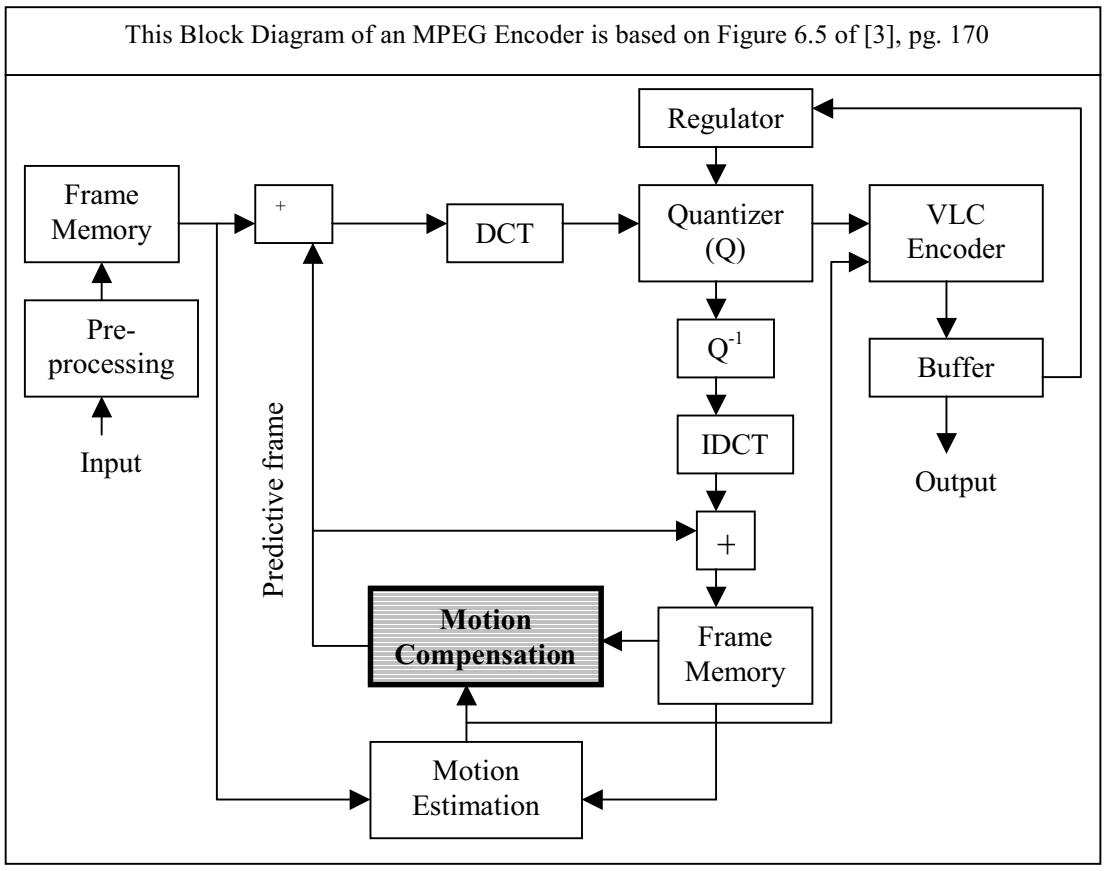

Figure 2: Block Diagram of an MPEG Encoder

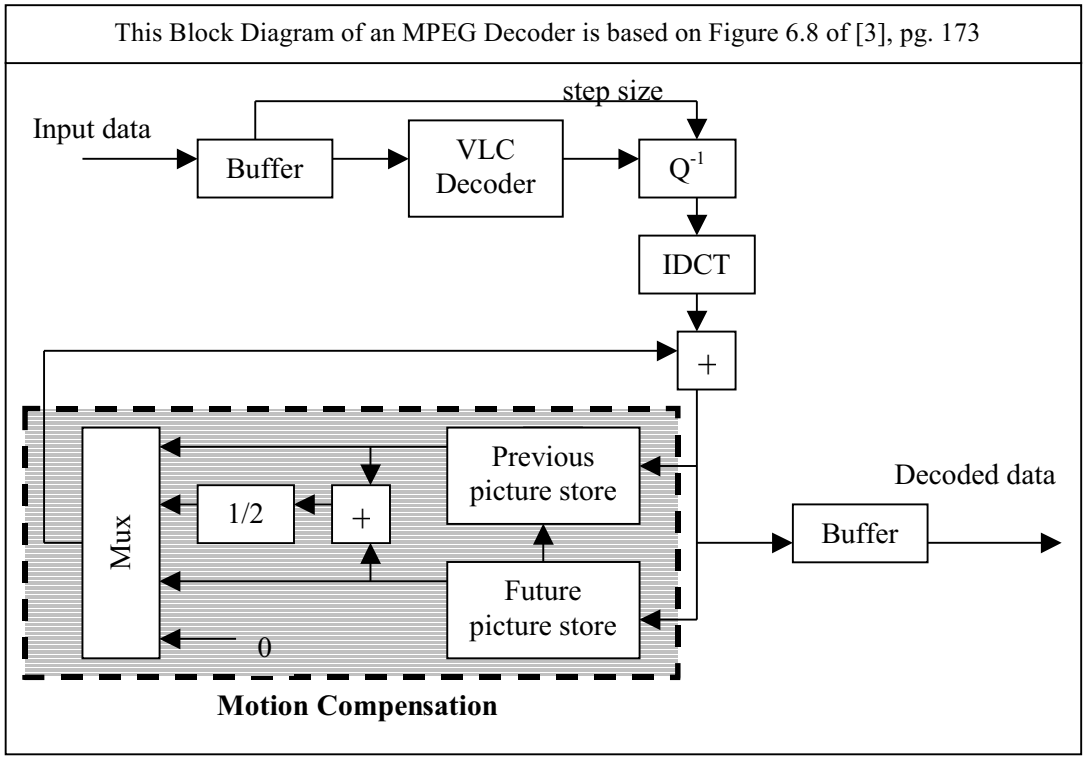

Figure 3: Block Diagram of an MPEG Decoder

2.2 Implementing the Motion Compensation Algorithm

The SSE2 instructions support SIMD operations on 16 bytes in parallel. This provides the fast data access MC requires.

In this application note we perform MC on the Y component only. A similar process can be used on the Cb and Cr 8x8 pel components.

The example presented here shows only the basic operations, not all the possible combinations for one and two reference frames with all the motion modes. It is easy to implement all the other modes based on the five examples shown here:

- Copying the 16x16 reference block to the currently built macroblock
- Averaging two horizontally adjacent pixels of the reference block
- Averaging two vertically adjacent pixels of the reference block
- Averaging four (2x2) neighboring pixels of the reference block
- Averaging two 2x2 pixel groups in two separate reference blocks that are then averaged together

2.2.1 Techniques for Instruction Selection and Accuracy

The packed averaging instruction `pavgb` enables a faster and more accurate MC algorithm than the algorithm presented in AP-529. The SSE2 instructions extend the `pavgb` instruction to support parallel averaging of 16 pixels (bytes).

The `pavgb` instruction averages two pixels exactly according to the MPEG standard: unsigned values are rounded up to the nearest integer. Motion modes that average two pixels using `pavgb` benefit from perfect accuracy and fast execution.

Averaging four pixels may produce an error of +1 when naively performing 3 average operations, i.e. `avg (a, b, c, d)=avg [avg (a,b),avg(c,d)]`. In the naïve implementation, the first stage performs two separate averages on the data elements `a,b,c,d` as follows: `avg(a,b)` and `avg(c,d)`. Each of these two first-stage averages can potentially contribute a +0.5 error to the final result. The second-stage average operation performs an average on the two results from the first stage as follows: `avg[avg(a,b), avg(c,d)]`. The second stage can potentially contribute a +0.5 error to the final result. The final result of the naïve operation therefore carries a potential error of +1; +0.5 from the first stage plus +0.5 from the second stage. For example, consider averaging the four values: 5, 2, 0, and 1. The correct result is 2, because `(5+2+0+1)/4=2`. However, when averaging according the above scheme, the calculation is: `avg[avg(5,2), avg(0,1)] = avg[4, 1] = 3`, that is a +1 error. Note that half integers are rounded up, so: `avg(5,2)=4` because `(5+2)/2=3.5`, `avg(0,1)=1`, and `avg(4,1)=3`.

Example 1 presents naïve code for averaging of four pixels.

```
movdqa xmm0, XMMWORD PTR [ a16 ]        ; load 16 pixels from src "a"
movdqa xmm1, XMMWORD PTR [ b16 ]        ; load 16 pixels from src "b"
movdqa xmm2, XMMWORD PTR [ c16 ]        ; load 16 pixels from src "c"
movdqa xmm3, XMMWORD PTR [ d16 ]        ; load 16 pixels from src "d"
pavgb  xmm0, xmm1  ; avg_1=simd_avg(a16,b16); // 1st stage averaging #1
pavgb  xmm2, xmm3  ; avg_2=simd_avg(c16,d16); // 1st stage averaging #2
```

Example 1: A Naïve 4-Pixel Averaging, Producing 16 Results in Parallel

The naïve code above generates an error of +1 in about 37% of the pixels. To minimize this error, subtract the value 1 from either of the two first stage averages, as presented in the next code example:

```
movdqa xmm0, XMMWORD PTR [ a16 ]         ; load 16 pixels from src "a"
movdqa xmm1, XMMWORD PTR [ b16 ]         ; load 16 pixels from src "b"
movdqa xmm2, XMMWORD PTR [ c16 ]         ; load 16 pixels from src "c"
movdqa xmm3, XMMWORD PTR [ d16 ]         ; load 16 pixels from src "d"
pavgb  xmm0, xmm1                        ; 1st stage averaging #1
pavgb  xmm2, xmm3                        ; 1st stage averaging #2
  ; avg_1 = sat_sub( avg_1, const_1_16_bytes )
psubusb xmm0, XMMWORD PTR [const_1_16_bytes] ; compensate errors
```

Example 2: More Accurate 4-Pixels Averaging, Producing 16 Results in Parallel

This simple and fast fix generates an error of -1 in only 13% of the pixels. This solution is used in the code listing of this application note. Obviously, the error distribution rate of the whole MC algorithm when using the other accurate modes is much lower.

Another alternative is to use fully accurate code for 4-pel averaging. This accurate (yet not optimized) code uses more computations, and is worth using if perfect accuracy is required. A version of this fully accurate code using the Intel® C++ SIMD Class Libraries for SIMD Operations for Intel® C/C++ Compiler is presented below:

```
Iu8vec16 Accurate_4pels_average ( Iu8vec16 a16, Iu8vec16 b16,
                                  Iu8vec16 c16, Iu8vec16 d16 )
{
   Iu8vec16 avg_1, avg_2, result;
   Iu8vec16 half_err_1st, half_err_2nd, fixup_mask;

   avg_1 = simd_avg(a16, b16);       // 1st stage averaging #1
   avg_2 = simd_avg(c16, d16);       // 1st stage averaging #2
   result = simd_avg(avg_1, avg_2);  // 2nd stage averaging

   // record 1 in lsb <=> some avg in the 1st stage carries +0.5 error
   half_err_1st = (a16 ^ b16) | (c16 ^ d16);
   // record 1 in lsb <=> 2nd stage avg carries at least +0.5 error
   half_err_2nd = ( avg_1 ^ avg_2);
   // record 1 <=> result carries +1 error
   fixup_mask = half_err_1st & half_err_2nd &(*(Iu8vec16*)const_1_16_bytes);

   return ( result - fixup_mask );   // compensate error where needed
```

Example 3: Accurate Averaging of 4 Pixels Using pavgb Instruction (SIMD width is 16)

3 Performance

Using the SSE2 instructions to implement the MC algorithm speeds up the performance relative to earlier technologies. This section describes the gains and improvements achieved using SSE2 instructions and addresses related programming considerations.

3.1 Gains/Improvements

The MC algorithm was previously optimized for MMX technology in AP-529. The SSE2 instructions are used to further optimize the implementation in this application note.

The speedup factors are:

- 128-bit SIMD integer instructions, enabling simultaneous processing of 16 pixels instead of 8
- fast and accurate `pavgb` instruction
- Pentium 4 processor micro-architecture improvements including:
  - — hardware prefetcher
  - — fast and wide data accesses to the caches and the memory
  - — improved branch prediction mechanism

3.2 Additional Considerations

3.2.1 Using Software Prefetch Instructions

The MC algorithm accesses data that is usually outside of the caches. In general, software prefetch instructions can be integrated into loops in order to minimize the memory latency effect. However, preliminary performance evaluation of the MC algorithm using a Pentium 4 processor did not show benefit from using prefetch instructions within the MC kernel.

This may be due to the fact that the Pentium 4 processor micro-architecture supports a hardware prefetch mechanism. Repeating patterns of fixed-stride consecutive load operations, as occur in the MC algorithm, are excellent candidates for benefiting from the Pentium 4 processor prefetch mechanism.

Another reason is that the calculations performed in the MC algorithm are too short to completely hide the prefetch latency. However, integrating prefetch instructions *outside* of the MC kernel can better hide the prefetch latency. For example, in the iDCT kernel, where calculations are heavy and memory traffic is low, prefetching can be beneficial to the overall performance. Similarly, applications that perform the decode/encode algorithms by batching several macroblocks together can benefit by integrating prefetches outside the MC kernel.

3.2.2 Aligned and Unaligned Data Accesses

Motion vectors can point to any address, so no alignment is guaranteed. The code example in this application note assumes that the motion vectors are all unaligned, and uses the unaligned load instruction `movdqu`.

An alternative method is to align the unaligned pointers. This method uses two aligned loads containing the required unaligned data, and uses "shift" and "or" instructions to extract the relevant data. This

method works well in cases where the exact shift amount is constant or at least confined to the high or low 64-bit register half. Motion vectors do not fall into this category.

For MC, the expense of shifting unaligned data outweighs the benefit of an aligned load.

4 Conclusion

The SSE2 instructions provide a significant performance gain in the MC algorithm for MPEG-2 decoding and encoding.

The following architecture improvements contribute to performance gain in MC:

- increased SIMD width
- the `pavgb` instruction

In addition, the following micro-architecture improvements contribute to performance gain:

- prefetch assistance
- improved branch prediction

5 C/C++ Coding Example

The following coding examples include five modes of the MC algorithm on Y component only.

```
#include <windows.h>
#include <dvec.h>
//-------------------------------------------------------------------------
// Global Declarations
_MM_ALIGN16 BYTE const_1_16_bytes[] =
                        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
//
// my_loadu() is used for unaligned loads, for more readable code than
// currently supported by Intel® C/C++ Compiler.
//
inline Iu8vec16 my_loadu(M128 *p)
{ return ( _mm_loadu_si128( (__m128i*)p ) ); }


//-------------------------------------------------------------------------
// Full motion (no average) from one frame only
//
void mc_WmtNI_full_b( DWORD stride, DWORD lines,
                      BYTE *dst_start, BYTE *bw_start )
{
    DWORD y;
    // num of 128 bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16  *src = (Iu8vec16*) bw_start;
    Iu8vec16  *dst = (Iu8vec16*) dst_start;

    // simply copy all the 16-pels lines from reference to dst
    for (y=0; y<lines; y++)
      dst[ y*stride_128 ] = my_loadu( (M128*)src + y*stride_128 );
 }
//-------------------------------------------------------------------------
// Average half-pels horizontally (in the "X" axis),
// from one reference frame only
//
void mc_WmtNI_hx_b( DWORD stride, DWORD lines,
```

```
                    BYTE *dst_start, BYTE *bw_start )
{
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;

    // avg each ref pixel with its adjacent pel, and store to dst
    for (y=0; y<lines; y++)
    {
      dst[ y*stride_128 ] = simd_avg(
        my_loadu( (M128*)( bw_start + y*stride ) ),
        my_loadu( (M128*)( bw_start + y*stride + 1 ) ) );
    }

}
//-------------------------------------------------------------------------
// Average half-pels vertically (in the "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hy_b( DWORD stride, DWORD lines,
                    BYTE *dst_start, BYTE *bw_start )
{
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD    stride_128 = stride >> 4;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;

    // avg each ref pixels with the pel from line below, and store to dst
    for (y=0; y<lines; y++)
    {
      dst[ y*stride_128 ] = simd_avg(
        my_loadu( (M128*)( bw_start + y*stride ) ),
        my_loadu( (M128*)( bw_start + (y+1)*stride ) ) );
    }
}
//-------------------------------------------------------------------------
// Average half-pels both horizontally and vertically ("X" and "Y" axis),
```

```
// from one reference frame only
//
void mc_WmtNI_hx_hy_b( DWORD stride, DWORD lines,
                       BYTE *dst_start, BYTE *bw_start )
{
    Iu8vec16 avg_above, avg_below;
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;

    // avg each 2x2 ref pixels, and store to dst
    for (y=0; y<lines; y++)
    {
      // avg 2 pels from upper line
      avg_above = simd_avg( my_loadu((M128*)(bw_start + y*stride)),
                            my_loadu((M128*)(bw_start + y*stride + 1)));
      // avg 2 pels from line below
      avg_below = simd_avg( my_loadu((M128*)(bw_start + (y+1)*stride)),
                            my_loadu((M128*)(bw_start + (y+1)*stride + 1)));
      // compensate possible error for 4-pels avg accuracy
      avg_above = sat_sub( avg_above, *(Iu8vec16*)const_1_16_bytes );
      // avg the two lines
      dst[ y*stride_128 ] = simd_avg( avg_above, avg_below );
    }
}
//-----------------------------------------------------------------------
// Average from two frames, each frame has 2x2 half-pels avg
//
void mc_WmtNI_hx_hy_b_hx_hy_f( DWORD stride, DWORD lines,
                               BYTE *dst_start, BYTE *fw_start, BYTE *bw_start )
{
    Iu8vec16 bw_avg, fw_avg, avg_above, avg_below;
    DWORD y;
    // num of 128bits (16 bytes) chunks are in stride
    DWORD stride_128 = stride >> 4;
    Iu8vec16 *dst = (Iu8vec16*) dst_start;
```

```
    for (y=0; y<lines; y++)
    {
      // 4-pels (2x2) avg from one reference frame
      avg_above = simd_avg( my_loadu((M128*)(bw_start + y*stride)),
                            my_loadu((M128*)(bw_start + y*stride + 1)));
      avg_below = simd_avg( my_loadu((M128*)(bw_start + (y+1)*stride)),
                             my_loadu((M128*)(bw_start + (y+1)*stride + 1)));
      avg_above = sat_sub( avg_above, *(Iu8vec16*)const_1_16_bytes );
      bw_avg = simd_avg( avg_above, avg_below );

      // 4-pels (2x2) avg from another reference frame
      avg_above = simd_avg( my_loadu((M128*)(fw_start + y*stride)),
                            my_loadu((M128*)(fw_start + y*stride + 1)));
      avg_below = simd_avg( my_loadu((M128*)(fw_start + (y+1)*stride)),
                            my_loadu((M128*)(fw_start + (y+1)*stride + 1)));
      avg_above = sat_sub( avg_above, *(Iu8vec16*)const_1_16_bytes );
      fw_avg = simd_avg( avg_above, avg_below );

      // avg result from both reference frame
      dst[ y*stride_128 ] = simd_avg( bw_avg, fw_avg );
    }
}
```

6 SSE2 Instructions Technology Assembly Code Example

The code in this section is identical code to the code in the previous section, except that it is written in assembly language containing SSE2 instructions.

```
//-----------------------------------------------------------------------
// Full motion (no average) from one frame only
//
void mc_WmtNI_full_b( DWORD stride, DWORD lines,
                      BYTE *dst_start, BYTE *bw_start )
{
    __asm
    {
      mov    edx, bw_start      ;
      mov    ecx, dst_start     ;
      mov    esi, lines         ;
      mov    eax, stride        ;
      lea    edi, [eax + eax]   ; stride*2

      ; simply copy all the 16-pels lines from reference to dst
      ;  each loop iteration copies 2 lines
nextLinesLoop:
      movdqu  xmm0, XMMWORD PTR [edx]     ; copy 2 lines from ref frame
      movdqu  xmm1, XMMWORD PTR [edx+eax] ;

      movdqa  XMMWORD PTR [ecx], xmm0     ; and store to dst lines
      movdqa  XMMWORD PTR [ecx+eax], xmm1 ;

      add    edx, edi                     ; advance 2 lines ahead
      add    ecx, edi
      sub    esi, 2
      jg      nextLinesLoop
    }
}

//-----------------------------------------------------------------------
// Average half-pels horizontally (in the "X" axis),
// from one reference frame only
//
```

```
void mc_WmtNI_hx_b( DWORD stride, DWORD lines,
                    BYTE *dst_start, BYTE *bw_start )
{
   __asm
   {
      mov    edx, bw_start      ;
      mov    ecx, dst_start     ;
      mov    eax, stride        ;
      mov    esi, lines         ;
      lea    edi, [eax + eax]   ; stride*2

      ; avg each ref pixel with its adjacent pel, and store to dst
      ;  each loop iteration computes 2 lines
nextLineLoop:
      movdqu xmm0, XMMWORD PTR [edx]      ; load 16 pels
      movdqu xmm1, XMMWORD PTR [edx+1]    ; and their adjacent pels
      movdqu xmm2, XMMWORD PTR [edx+eax]  ; do the same for line below it
      movdqu xmm3, XMMWORD PTR [edx+eax+1]

      pavgb  xmm0, xmm1                   ; avg one line
      pavgb  xmm2, xmm3                   ; avg second line

      movdqa  XMMWORD PTR [ecx], xmm0     ; and store the results
      movdqa  XMMWORD PTR [ecx+eax], xmm2

      add    edx, edi                     ; advance 2 lines ahead
      add    ecx, edi
      sub    esi, 2
      jg     nextLineLoop
   }
}

//--------------------------------------------------------------------------
// Average half-pels vertically (in the "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hy_b( DWORD stride, DWORD lines,
                    BYTE *dst_start, BYTE *bw_start )
```

```
{
    __asm
    {
      mov    edx, bw_start      ;
      mov    ecx, dst_start     ;
      mov    eax, stride        ;
      mov    esi, lines         ;
      lea    edi, [eax + eax]   ; stride*2

      ; avg each ref pixels with the pel from line below, and store to dst
      ;  each loop iteration computes 2 result lines

      movdqu  xmm0, XMMWORD PTR [edx]           ; prepare data for 1st iter
nextLineLoop:
      ; xmm0 already holds the first line pels
      movdqu  xmm1, XMMWORD PTR [edx+eax]       ; load 2nd line
      movdqu  xmm2, XMMWORD PTR [edx+edi]       ; and 3rd line

      pavgb  xmm0, xmm1                         ; avg 1st line with 2nd
      pavgb  xmm1, xmm2                         ; avg 2nd line with 3rd

      movdqa  XMMWORD PTR [ecx], xmm0     ; store result to first line
      movdqa  xmm0, xmm2                  ; save "first" line of next iter
      movdqa  XMMWORD PTR [ecx+eax], xmm1 ; store result to second line

      add     edx, edi                    ; advance 2 lines ahead
      add     ecx, edi
      sub     esi, 2
      jg      nextLineLoop
    }
}
//-----------------------------------------------------------------------
// Average half-pels both horizontally and vertically ("X" and "Y" axis),
// from one reference frame only
//
void mc_WmtNI_hx_hy_b( DWORD stride, DWORD lines,
                       BYTE *dst_start, BYTE *bw_start )
{
```

```
    __asm
    {
      mov    edx, bw_start      ;
      mov    ecx, dst_start     ;
      mov    eax, stride        ;
      mov    esi, lines         ;
      lea    edi, [eax + eax]   ; stride*2

      ; avg each 2x2 ref pixels, and store to dst

      movdqa xmm7, XMMWORD PTR [const_1_16_bytes] ; used for higher accuracy

      movdqu  xmm0, XMMWORD PTR [edx]           ; prepare data for 1st iter
      movdqu  xmm1, XMMWORD PTR [edx+1]         ; prepare data for 1st iter
nextLineLoop:
      ; xmm0 already holds the first line pels
      ; xmm1 already holds the first line adjacent pels
      movdqu  xmm2, XMMWORD PTR [edx+eax]       ; load 2nd line
      movdqu  xmm3, XMMWORD PTR [edx+eax+1]     ; and its adjacent pels
      movdqu  xmm4, XMMWORD PTR [edx+edi]       ; load 3rd line
      movdqu  xmm5, XMMWORD PTR [edx+edi+1]     ; and its adjacent pels

      pavgb  xmm0, xmm1         ; horizontal avg of 1st line
      pavgb  xmm2, xmm3         ; horizontal avg of 2nd line
      movdqa xmm1, xmm5         ; save "first" adjacent line for next iter
      pavgb  xmm5, xmm4         ; horizontal avg of 3rd line

      psubusb xmm2, xmm7        ; compensate error for accuracy

      pavgb  xmm0, xmm2         ; vertical avg for 1st dst line
      pavgb  xmm2, xmm5         ; vertical avg for 2nd dst line

      movdqa  XMMWORD PTR [ecx], xmm0     ; store 1st line to dst
      movdqa xmm0, xmm4                   ; save "first" line for next iter
      movdqa  XMMWORD PTR [ecx+eax], xmm2  ; store 2nd line to dst

      add    edx, edi           ; advance 2 lines ahead
      add    ecx, edi
```

```
      sub    esi, 2
      jg      nextLineLoop
   }
}
//-----------------------------------------------------------------------
// Average from two frames, each frame has 2x2 half-pels avg
//
void mc_WmtNI_hx_hy_b_hx_hy_f( DWORD stride, DWORD lines,
                           BYTE *dst_start, BYTE *fw_start, BYTE *bw_start )
{
   __asm
   {
      mov    edx, bw_start      ;
      mov    edi, fw_start      ;
      mov    ecx, dst_start     ;
      mov    eax, stride        ;
      mov    esi, lines         ;

      movdqa xmm7, XMMWORD PTR [const_1_16_bytes] ; used for higher accuracy

      movdqu  xmm0, XMMWORD PTR [edx]     ; prepare data for 1st iter
      movdqu  xmm1, XMMWORD PTR [edx+1]

      movdqu  xmm4, XMMWORD PTR [edi]
      movdqu  xmm5, XMMWORD PTR [edi+1]

nextLineLoop:
      ; xmm0 already holds the first bw line pels
      ; xmm1 already holds the first bw line adjacent pels
      movdqu  xmm2, XMMWORD PTR [edx+eax]       ; load 2nd bw line
      movdqu  xmm3, XMMWORD PTR [edx+eax+1]     ; and its adjacent pels

      ; xmm4 already holds the first fw line pels
      ; xmm5 already holds the first fw line adjacent pels
      movdqu  xmm6, XMMWORD PTR [edi+eax]       ; load 2nd fw line

      pavgb  xmm0, xmm1     ; horizontal avg of 1st bw line
      movdqa xmm1, xmm3     ; save "first" adjacent bw line for next iter
```

```
      pavgb   xmm3, xmm2     ; horizontal avg of 2nd bw line

      psubusb xmm3, xmm7     ; compensate error for accuracy

      pavgb   xmm3, xmm0     ; vertical avg for bw line
      movdqa  xmm0, xmm2     ; save "first" bw line for next iter

      movdqu  xmm2, XMMWORD PTR [edi+eax+1]     ; load 2nd fw line (adjacent)
      pavgb   xmm4, xmm5     ; horizontal avg of 1st fw line
      movdqa  xmm5, xmm2     ; save "first" adjacent fw line for next iter
      pavgb   xmm2, xmm6     ; horizontal avg of 2nd fw line

      psubusb xmm2, xmm7     ; compensate error for accuracy

      pavgb   xmm2, xmm4     ; vertical avg for fw line
      movdqa   xmm4, xmm6    ; save "first" fw line for next iter

      pavgb   xmm3, xmm2     ; final result: avg bw and fw frames

      movdqa   XMMWORD PTR [ecx], xmm3   ; store to dst

      add     edx, eax       ; advance 1 line ahead
      add     edi, eax
      add     ecx, eax
      sub     esi, 1
      jg       nextLineLoop
   }
}
```

Appendix A - Performance Data

Performance Data Revision History

Revision	Revision History	Date
2.0	Update for Pentium 4 processor 1.2 GHz	07/00
1.0	Original publication of document	9/99

The code samples provided with this application note provide three different implementations of the Motion Compensation function:

1. Intel® C++ SIMD Class Libraries for SIMD Operations (IVEC).
2. Assembly language using Streaming SIMD Extensions 2 (SSE2) Instructions.
3. C++ SIMD classes - SSE2 Instructions (DVEC).

Performance results have not been calculated for the third version at this time.

Table 1: Performance Data of Motion Compensation (MC) Implementations (cold cache)

Performance Data in Microseconds per Iteration - cold cache				
	Intel Pentium III Processor		Intel Pentium 4 Processor	
	full_b	hx_hy_b_hx_hy_f	full_b	hx_hy_b_hx_hy_f
SSE IVEC	2.58	5.17	1.54	3.39
SSE2 ASM	-	-	1.48	3.24

Table 2: Speedups from Table 1 Performance Data (cold cache)

Implementations and Platforms	Speedup full_b	Speedup hx_hy_b_hx_hy_f
Pentium 4 Processor (SSE2 ASM vs. SSE IVEC)	1.04	1.05
SSE IVEC (Pentium 4 processor vs. Pentium III processor)	1.67	1.52
SSE2 ASM on Pentium 4 processor vs. SSE on Pentium III processor	1.74	1.60

Table 3: Performance Data of MC Implementations (perfect cache)

Performance Data in Microseconds per Iteration - perfect cache				
	Pentium III Processor		Pentium 4 Processor	
	full_b	hx_hy_b_hx_hy_f	full_b	hx_hy_b_hx_hy_f
SSE IVEC	0.434	1.19	0.373	0.990
SSE2 ASM	-	-	0.260	0.838

Table 4: Speedups from Table 3 Performance Data (perfect cache)

Implementations and Platforms	Speedup full_b	Speedup hx_hy_b_hx_hy_f
Pentium 4 processor (SSE2 ASM vs. SSE IVEC)	1.435	1.181
SSE IVEC (Pentium 4 processor vs. Pentium III processor)	1.16	1.20
SSE2 ASM on Pentium 4 processor vs. SSE on Pentium III processor	1.67	1.42

Performance was measured using a synthetic test app, with random motion vectors, both on "cold cache" environment, by trashing the caches before each frame, and on “perfect cache” environment. The performance was measured with a 733 MHz Pentium III processor and a 1.2 GHz Pentium 4 processor. See Test Systems Configuration on page A-3, for a detailed description of the two systems used for measuring.

As can be seen in the "cold cache" environment in **Table 2**, the SSE2 implementation of the `hx_hy_b_hx_hy_f` motion mode is 1.05 times faster than the SSE implementation when both implementations are executed on a Pentium 4 processor.

The "perfect cache" environment in **Table 4** introduces better performance results, by canceling memory effects. The SSE2 implementation of the `hx_hy_b_hx_hy_f` motion mode is 1.181 times faster than the SSE implementation when both implementations are executed on a Pentium 4 processor.

The `hx_hy_b_hx_hy_f` mode is the heaviest mode in computations and in memory bandwidth. The `full_b` mode has no computation operations at all, and it only copies memory, it introduces speedup of 1.04 times faster in "cold cache" environment, and a speedup of 1.435 in "perfect cache" environment. The rest of the motion modes presented in the application note document use less computation operations than `hx_hy_b_hx_hy_f`, with same memory traffic as `full_b.` They were not measured for performance.

The above speedup is attributed to the following optimizations:

- Larger SIMD width. The integer SSE2 instructions use the 128-bit XMM registers instead of the 64-bit MMX technology registers. The increased SIMD width decreases register pressure and doubles the amount of data processed per instruction.
- Both the SSE2 implementation and the SSE implementation take advantage of the `pavgb` instruction, which fasten them over the MMX technology code presented in AP-529. The SSE implementation on Pentium III processor takes advantage of software prefetching by using the `prefetch` instruction.

Test Systems Configuration

Table 5: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 6: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195